Arguing satisfaction of security requirements

Book Section

Arguing Satisfaction of Security Requirements

Authors:

Charles B. Haley	C.B.Haley@open.ac.uk
Robin Laney	R.C.Laney@open.ac.uk
Jonathan D. Moffett	J.Moffett@open.ac.uk
Bashar Nuseibeh	B.Nuseibeh@open.ac.uk

The postal address for all authors is:

Department of Computing
The Faculty of Mathematics and Computing
The Open University
Walton Hall
Milton Keynes
Buckinghamshire
MK7 6AA
UK

Telephone and fax for all authors is:

Tel: +44 1908 653037
Fax: +44 1908 652335

In addition, Charles Haley (the corresponding author) can be reached at

Tel: +44 1908 858825
Fax: +44 70 75055128

Keywords: security requirements engineering, design rationale, argumentation, system context

Arguing Satisfaction of Security Requirements

Abstract

The chapter presents a process for security requirements elicitation and analysis, based around the construction of a satisfaction argument for the security of a system. The process starts with the enumeration of security goals based on assets in the system, then uses these goals to derive security requirements in the form of constraints. Next, a satisfaction argument for the system is constructed, using a problem-centered representation, a formal proof to analyze properties that can be demonstrated, and structured informal argumentation of the assumptions exposed during construction of the argument. Constructing the satisfaction argument can expose missing and inconsistent assumptions about system context and behavior that effect security, and a completed argument provides assurances that a system can respect its security requirements.

INTRODUCTION

This chapter describes an approach to carry out security requirements engineering; the process of eliciting, specifying, and analyzing the security requirements for a system. The approach is founded on the following fundamental ideas:

- The "what" of security requirements – its core artifacts – must be understood before the "how" of their construction and analysis.
- Security cannot be considered as a feature of software alone; it is concerned with the prevention of harm in the real world. We must therefore consider both the security requirements of real-world systems and the specification of software that demonstrably meets those requirements.
- Since security is largely concerned with prevention of misuse of system functions, security

requirements can most usefully be defined by considering them as constraints upon functional requirements.

- Since security is by definition an 'open world' problem (the domain of analysis will always be too small), any argument that a system will satisfy its security requirements must take non-provable assertions about the real world into account.

The contribution of this chapter is the combination of four components described in previous work into a coherent security requirements process. The first component is a framework that provides a systematic statement of the roles and relationships of security goals, security requirements, and security functions, and their relationships with other system and software requirements. The second is a way of describing threats and their interactions with the system. The third is a precise definition of security requirements. The fourth is a two-layer set of arguments to assist with validating the security requirements within the context of the system, to determine that the system is able to meet the security requirements placed upon it. The first and third were described in (Moffett, Haley, & Nuseibeh, 2004), the second in (Haley, Laney, & Nuseibeh, 2004), and the fourth in (Haley, Moffett, Laney, & Nuseibeh, 2005).

Although all the steps in the process are described in this chapter, the third component – argumentation – is emphasized. This is not an exclusive focus, though, as understanding the role of argumentation in security requirements requires that one understand the first three parts of the process in order to place the pieces correctly in context.

THE SECURITY REQUIREMENTS PROCESS

The process uses the framework described in (Moffett et al., 2004), which enables an understanding of the role of requirements analysis in the validation and verification of security

requirements and the other artifacts in the framework. The framework integrates the concepts of the two disciplines of requirements engineering and security engineering. From requirements engineering it takes the concept of functional goals, which are operationalized into functional requirements, with appropriate constraints. From security engineering it takes the concept of assets, together with threats of harm to those assets. In the framework:

- Security goals aim to protect assets from security threats.
- Security goals are operationalized into primary security requirements, which take the form of (a subset of) the constraints on the functional requirements.
- Primary security requirements are realized by requirements on the behavior of relevant domains, and in particular the domain of software.
- Feasible realizations of the primary security requirements lead to the need for secondary security requirements, which are additional required functions or constraints.

The framework has been developed in order to understand the place of security requirements within the development of an individual application, and our proposals are limited by that scope. The application will, of course, be developed in the context of a software operating environment, a hardware environment, and a human cultural environment. All of these environments will have properties which impact upon the application.

The Need for Defined Security Requirements

Consider Anderson's report (Anderson, 1996), which presents a view of the security goals of a hospital clinical information system from the point of view of the doctors. It makes explicit assumptions that the doctors should have control of the system, while the administrators should be subordinate. In reality, in many health services, there is a power struggle between doctors and

administrators. In a hypothetical system in which that power struggle has not been resolved, we can consider two possible sets of candidate security requirements. In set 1, proposed by the doctors, some actions are considered legitimate for doctors, but prohibited for administrators. In set 2, proposed by the administrators, the situation is reversed.

However, the system's requirements in this instance must be free of conflicts, because if not, the implementers may resolve the conflicts in potentially inconsistent and incorrect ways. The conflict between rival stakeholders must be resolved by the production of an agreed set of security requirements. Only then can we analyze the requirements for misuses or abuses.

Definition of Security Requirements

We define the **primary security requirements** of a system as *constraints on the functions of the system*: the security goals are operationalized as constraints on the system's functional requirements:

- They are constraints on the system's functional requirements, rather than themselves being functional requirements.
- They express the system's security goals in operational terms, precise enough to be given to a designer.

Secondary security requirements are additional required functions or constraints that are derived in the course of requirements analysis or system design, in order to enable feasible realizations of the primary security requirements

It is worth noting that we do not claim to be *correct* in defining primary security requirements as constraints on functional requirements, simply that it is *useful* to do so. We can be pragmatic in this way because:

- Requirements specifications, in general, describe the functions (or operations or services) to be provided by a system.
- It is clearly desirable for the specification to describe security requirements in a way that enables them immediately to be related to the functions.

It is important to reiterate that security requirements (constraints) are expressed in terms of the system context, which is larger than the software. A constraint on a function can be realized in multiple ways, some completely outside the software to be constructed. We use a variant of Jackson's problem frames (Jackson, 2001) to represent the system context. The reader will find further detail in the section *Problem Frames* later in this chapter.

Validation by Argumentation

One key validation step for the process described in this chapter is the ability to show that the system can satisfy the security requirements. We propose the use of structured informal and formal argumentation for this validation step: to convince a reader that a system can satisfy the security requirements laid upon it. These arguments, called *satisfaction arguments*, are in two parts. The first part consists of a formal argument to prove a system can satisfy its security requirements, drawing upon claims about a system, and assuming the claims are accepted. The second part consists of structured informal arguments to support the claims about system behavior and characteristics that were made in the first argument. Building on our understanding of security requirements, two-step satisfaction arguments assist with determining security-relevant system properties, and how inconsistent and implausible assumptions about them affect the security of a system.

The Resulting Artifacts

We assume that the system development process has recognizable stages, each of which

produces artifacts that are successively closer representations of a working system. These are **core artifacts**. They are ordered in an abstraction hierarchy, progressing from the most abstract to the final concrete working system. At early stages, core artifacts are typically documents or prototypes. The final core artifact is the working system itself, consisting of a combination of physical and software items.

There are two sets of core artifacts in which we have most interest. On the mainstream requirements engineering side, we find goals, requirements, and the system (not software) architecture. On the security engineering side we find assets, threats and control principles.

Support artifacts are artifacts that help to develop, analyze or justify the design of a core artifact. They may represent formal analysis, informal argument, calculation, example or counter-example, etc. They are the by-products of processes whose aim is to help produce verified and valid core artifacts: either constructive processes which help create them, or analytical processes which test them, both internally (verified) and in relation to their senior artifacts in the hierarchy (valid).

Dependencies between Artifacts. In a hierarchy of artifacts, there are dependencies between artifacts. For example, an operationalized requirement is dependent upon a higher-level goal from which it has been derived, because alteration of the goal may cause alteration of the requirement. We will call this kind of dependency **hierarchical dependency**.

There is also a reverse kind of dependency: **feasibility**. If it proves impossible to implement a system that completely satisfies a requirements specification, then this will force a change in the goals or requirements. The higher-level artifact is dependent on the feasibility of the artifacts below it in the hierarchy.

The dependency relationships have an important implication for the structure of development

processes: if an artifact is dependent upon the implementation of another artifact for its feasibility, then if the implementation is not feasible, there must be an iteration path in the process, back to the ancestor from its descendant.

RELATED WORK

Our work is related to, and builds upon, research on security requirements, design rationale and argument capture, safety requirements analysis.

Security Requirements

We review previous work on security requirements by examining security goals, security policies, non-functional requirements, and other definitions. Before discussing these it may be useful to point out what security requirements are *not*.

Security Requirements are not Security Functions. Security requirements are not to be identified with security functions; e.g., encryption, access control, etc. Howard (Howard & LeBlanc, 2001) states this in the following terms:

```
Security Features != Secure Features
```

The provision of security functionality in a system is only useful if it supports defined and well understood security objectives.

Security Requirements as Security Goals. Many authors implicitly assume that security requirements are identical to security goals. Tettero et al (1997) are explicit about this, defining security requirements as the confidentiality, integrity, and availability of the entity for which protection is needed. While this is a clear definition, it is too abstract for many purposes: no doubt both doctors and the administrators in the example previously presented would agree on the importance of confidentiality, integrity, and availability of the clinical information, but they

would disagree on the concrete security requirements that express those goals. If designers were only given security goals to work with, it would be necessary for them to carry out further work that belongs in the domain of the requirements engineer, by deciding how the security goals should be operationalized in the requirements.

Security Requirements as Security Policies. Some authors identify security requirements with security policies. Devanbu & Stubblebine (2000) define a security requirement as "a manifestation of a high-level organizational policy into the detailed requirements of a specific system. [… We] loosely (ab)use the term 'security policy' […] to refer to both 'policy' and 'requirement'". Anderson (2001) is less direct; he states that a security policy is "a document that expresses […] what […] protection mechanisms are to achieve" and that "the process of developing a security policy […] is the process of requirements engineering".

The difficulty with security policies is their chameleon-like meaning; the term can be used for anything from a high-level aspiration to an implementation. Therefore, without accompanying detailed explanation, it is not satisfactory to define security requirements as security policies.

Security Requirements as Non-functional Requirements. Devanbu & Stubblebine (2000), in addition to their definition above, remark that security requirements are a kind of non-functional requirement. We agree with that comment, but it needs further explanation. Kotonya and Sommerville (1998), when discussing non-functional requirements, in which they include security, define them as "restrictions or constraints" on system services; similar definitions can be found in other text books. Rushby (2001) appears to take a similar view, stating "security requirements mostly concern what must *not* happen". Using the Tropos methodology, Mouratidis et al (2003) state that "security constraints define the system's security requirements".

Our own view is consistent with these definitions: that security requirements are most usefully

defined as requirements for constraints on system functions.

Other Definitions of Security Requirements. Lee et al (2002) point out the importance of considering security requirements in the development life cycle, but do not define them. ISO/IEC 15408 (ISO/IEC, 1999) does not define security requirements in its glossary. However, in one place they are depicted as being at a higher level than functional requirements, but elsewhere the reference to "security requirements, such as authorization credentials and the IT implementation itself" appears to us as being at too *low* a level! Heitmeyer (2001) shows how the SCR method can be used to specify and analyze security properties, without giving the criteria for distinguishing them from other system properties.

A number of papers have focused on security requirements by describing how they may be violated. For example, McDermott & Fox (1999), followed independently by Sindre & Opdahl (2000) and elaborated by Alexander (2003), describe abuse and misuse cases, extending the use case paradigm to undesired behavior. Liu et al (2003) describe a method of analyzing possible illicit use of a system, but omit the important initial step of identifying the security requirements of the system before attempting to identify their violations.

Van Lamsweerde (2004) describes a process by which security goals are made precise and refined until reaching security requirements; he does not appear to regard them as different from any other kind of requirement. Antón & Earp (2001) use the GBRAM method to operationalize security goals for the generation of security policies and requirements, but do not define security requirements.

Design Rationale and Argument Capture

Design rationale is principally concerned with capturing how one arrived at a decision, alternate decisions, or the parameters that went into making the decision (J. Lee & Lai, 1991).

For example, Buckingham Shum (2003) focuses on how rationale (argument) is visualized, especially in collaborative environments. Potts and Bruns (1988), and later Burge and Brown (2004) discuss capturing how decisions were made, which decisions were rejected, and the reasons behind these actions. Mylopoulos et al (1990) present a way to represent formally knowledge that was captured in some way, without focusing on the outcome of any decisions. Ramesh and Dhar (1992) describe a system for "capturing history in the upstream part of the life cycle." Fischer et al (1996) suggest that the explicit process of argumentation can itself feed into and benefit design. Finkelstein and Fuks (1989) suggest that the development of specifications by multiple stakeholders, who hold disparate views, may be achieved through an explicit dialogue that captures speech acts, such as assertions, questions, denials, challenges, etc. The representation of the dialogue is then a rationale for the specifications constructed. The common element in all of the above work is the capture over time of the thoughts and reasons behind decisions. Whether the decisions satisfy the needs is not the primary question.

Safety cases

Kelly (1999) argues that "a safety case should communicate a clear, comprehensive and defensible argument that a system is acceptably safe to operate in a particular context." He goes on to show the importance of the distinction between *argument* and *evidence*. An argument calls upon appropriate evidence to convince a reader that the argument holds. Attwood et al use the same principles in (2004), taking the position that argument is a bridge between requirements and specification, permitting capture of sufficient information to realize rich traceability. Our work combines these two ideas.

The techniques proposed by Kelly are not directly applicable to security without modification, primarily because the techniques are focused around objective evidence, component failure, and

accident, rather than subjective reasoning, subversion, and malicious intent.

ARGUMENTATION DRIVEN PROBLEM ANALYSIS

We use Zave and Jackson's approach to problem analysis (Jackson, 2001; Zave & Jackson, 1997). They argue that one should construct a correctness argument for a system, where the argument is based on known and desired properties of the domains involved in the problem. To quote Jackson, "Your [correctness] argument must convince yourself and your customer that your proposed machine will ensure that the requirement is satisfied in the problem domain."

This section summarizes our approach to problem analysis using problem frames, and describes the two types of arguments. We begin with a short presentation of problem frames.

Problem Frames

The view of requirements exemplified by problem frames (Jackson, 2001) is that a system is intended to solve a *problem* in a *context* of *real-world physical domains*, where the context includes system design decisions. One uses problem frames to analyze the problem in terms of the context and the design decisions the context represents. The context contains domains, which are physical elements around which the system (not just the software) will be built. The problem frames approach differs from some other approaches (e.g. KAOS (van Lamsweerde, 2001)) that hold that a requirements engineer should reason about a system's characteristics without using a physical model of the world; under this view, a requirements engineer enumerates goals for a system under consideration and produces a temporal logic model of the system's desired behavior. We will show how using the real-world system perspective provided by problem frames assists with the determination of security requirements.

In the problem frames universe, all computing problems involve the interaction between

domains in the world. Domains are tangible (e.g. people, equipment, networks) but may contain intangibles (e.g. information). Every domain has *interfaces* to other domains, which are defined by the *phenomena* visible between the domains on the interfaces. Descriptions of phenomena controlled by given (existing) domains are *indicative*; they are "objectively true" (Jackson, 2001), meaning the phenomena and resulting behavior can be observed. Descriptions of phenomena of designed domains (domains to be built as part of the solution) are *optative*; one hopes to observe the phenomena in the future.

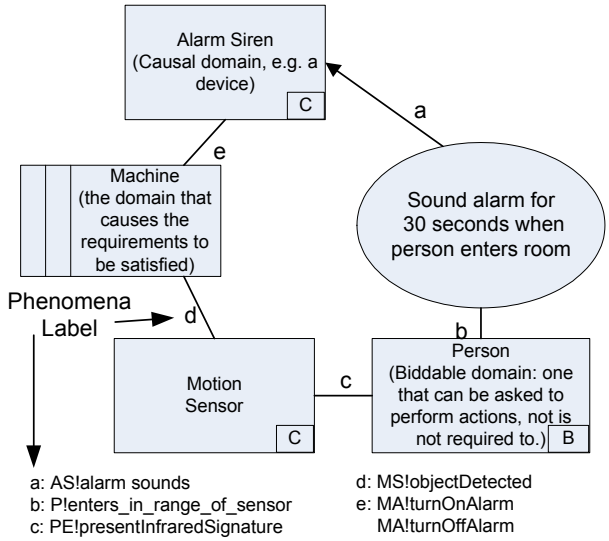

Figure 1 – Example Problem Diagram

Problems are described using a straightforward graphical notation. Domains are boxes, interfaces are arrows, and phenomena are described by indicated by a label on an interface. Figure 1 shows a problem diagram for a simple alarm system. It has one requirement: sound an alarm for 30 seconds when a person enters the room under protection.

One special domain is the *machine*, the domain that performs the transformations to satisfy the *requirement*. The machine is indicated by 2 vertical lines in the domain box. The interplay of phenomena between the machine and its connected domains defines what the machine has to

work with to satisfy the requirement. The interplay of phenomena is a *specification*, describing how the requirements are satisfied (Zave & Jackson, 1997). The difference between specification and requirement is important. A specification is an expression of the behavior of phenomena visible at the boundary of the domains, whereas a requirement is a description of the problem to be solved. For example, in the context of a building we might find the requirements 'permit passage from one room to another' and 'physically separate rooms when possible'. Clearly the problem involves something like doors. Equally as clearly, it does not specify that doors be used, nor does it specify internal phenomena or behavior. It is up to the designer (the architect in this case) to choose the 'door' domain(s) for the system. One might satisfy the requirement with a blanket, an automatic door, a futuristic iris, or a garden maze. Each domain implementation presents different phenomena at its boundaries (i.e. they work differently), and the resulting system specification must consider these differences. However, the requirement does not change.

There are two fundamental diagram types in a problem frames analysis, *context diagrams* and *problem diagrams*. A context diagram shows the domains of interest in a system, how the domains are interconnected, and optionally the phenomena on the interfaces between domains. A problem diagram is used to describe a *problem* in the system; the problem is expressed by a requirement. The problem diagram, of which Figure 1 is an example, is a projection of the context, showing only the domains or groups of domains of interest to the particular problem.

The diagram shows the domains involved and the phenomena exchanged between the domains. It shows the requirement (the function), the constrained domain(s), the inputs, and the phenomena shared between the domains: the domains that are involved in the system within which the machine operates to realize the necessary function. The *behavior* of the system is specified by the sequencing and interplay of phenomena between the domains. Behavior is used

to construct the correctness argument by making *claims* about the behavior. To ground the idea of 'claim' in Jackson's problem analysis, system requirements are optative statements, or statements about what we wish to be true, about the behavior of a system, and therefore are claims about future system behavior that should be argued (and in fact, this is what correctness arguments do). For example, the optative statement "the system shall do X" states a claim that under the conditions described in the problem, the system *will* do X. The correctness argument establishes the validity of this claim.

A similar situation exists with regard to security requirements and correctness arguments. Two significant distinctions must be made, however. The first is that it is very difficult to talk about correctness when discussing security. One can convince the reader that the proposed system meets the needs, but it is far more difficult to prove that the system is correct. The distinction between convince and prove (or show) is important. It is not possible to prove the negative – that violation of security goals do not exist – but one can be convincing that sufficient outcomes have been addressed. We propose using argumentation to this end: to convince a reader that the security requirements can be satisfied.

Trust Assumptions & Arguments

Our earlier work extended the problem frames approach with *trust assumptions* (Haley, Laney, Moffett, & Nuseibeh, 2004), which are claims about the behavior or the membership of domains included in the system, where the claims are made in order to satisfy a security requirement. These claims represent an analyst's trust that domains behave as described. Trust assumptions are in the end the analyst's opinion, and therefore assumed to be true. Said another way, trust assumptions are unsubstantiated claims used in security satisfaction arguments.

Any form of security argument must satisfy two goals: 1) that given a collection of domain

properties and trust assumptions, one can show that a system can be secure, and 2) to create a uniform structure for the satisfaction argument so that the trust assumptions are made explicit. We satisfy these goals by splitting the satisfaction argument into two parts: a *formal outer argument* that is first constructed, and *informal structured inner arguments* that are constructed to support the outer argument. The inner argument makes extensive use of trust assumptions.

The Outer Argument

The formal outer argument uses claims about the behavior of the system (interplay of phenomena) to demonstrate that the security requirement (the constraint) is satisfied. It is expressed using an appropriate logic, where the premises are formed from domain behavior properties and the conclusion is the satisfaction of the security requirement. For simplicity, we use propositional logic in this chapter, resulting in the outer argument being a proof of the form:

```
(domain property premises) ⊢ security requirement
```

The Inner Arguments

The inner argument is a set of informal arguments to recursively support the claims used in the outer argument. We propose a form inspired by the work of Toulmin (1958), one of the earliest advocates and developers of a formal structure for human reasoning. Toulmin style arguments appear to be well suited for our purpose, since they facilitate the capture of relationships between domain properties (grounds in the formal argument), the trust assumptions that eventually support these grounds, and reasons why the argument may not be valid.

Toulmin et al (1979) describe arguments as consisting of:

1. *Claims*, specifying the end point of the argument – what one wishes to convince the world of.
2. *Grounds*, providing any underlying support for the argument, such as evidence, facts, common knowledge, etc.
3. *Warrants*, connecting and establishing relevancy between the grounds and the claims. A

warrant explains how the grounds are related to the claim, not the validity of the grounds themselves.

4. *Backing*, establishing that the warrants are themselves trustworthy. These are, in effect, grounds for believing the warrants.
5. *Modal qualifiers*, establishing within the context of the argument the reliability or strength of the connections between warrants, grounds, and claims.
6. *Rebuttals*, describing what might invalidate any of the grounds, warrants, or backing, thus invalidating the support for the claim.

Toulmin et al (1979) summarize the above six items as follows: "The *claims* involved in real-life arguments are, accordingly, *well founded* only if sufficient *grounds* of an appropriate and relevant kind can be offered in their support. These grounds must be connected to the claims by reliable, applicable, *warrants*, which are capable in turn of being justified by appeal to sufficient *backing* of the relevant kind. And the entire structure of argument put together out of these elements must be capable of being recognized as having this or that kind and degree of certainty or probability as being dependent for its reliability on the absence of certain particular extraordinary, exceptional, or otherwise *rebutting* circumstances." They propose a diagram for arguments that indicates how the parts fit together. See Figure 2.

Newman and Marshall (1991) show that the 'pure' Toulmin form suffers because the fundamental recursive nature of the argument is obscured. Grounds may need to be argued, making them claims. Warrants may need to be argued, which is the reason for the existence of the backing, but it is not clear how the backing differs from grounds in a normal argument. We agree, and extend Toulmin arguments to make the recursive properties of arguments and the relationships between grounds, warrants, and claims explicit, while keeping the basic

connections between the components that Toulmin proposed.

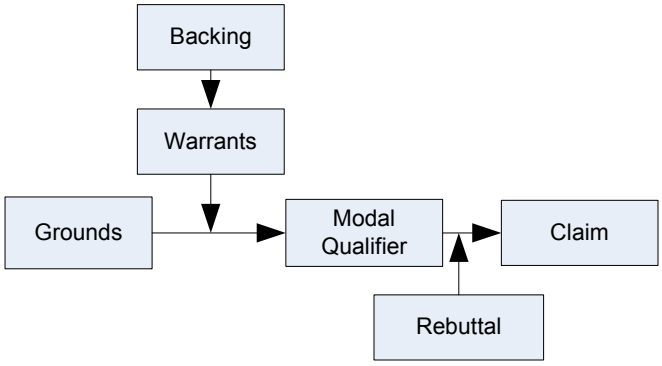

Figure 2 – Generic Toulmin-form argument

We propose a simple language to represent the structure of these extended Toulmin arguments. The language, with a syntax formally defined by an LR(1) grammar, captures the essence of Toulmin arguments while facilitating recursion and sub-arguments. We chose a textual language because a) textual utterances are easier to manipulate than tree diagrams, b) trees are easily generated from the parser's abstract syntax tree, and c) a 'compiler' can assist in dynamic browsing of arguments. Further discussion of the use of the language can be found in the case study.

FROM SECURITY GOALS TO SECURITY REQUIREMENTS

This section presents the 3 major steps of our process: identification of security goals, identification of security requirements, and construction of satisfaction arguments.

Process Overview

Our process for moving from security goals to security arguments is shown as Figure 3. There are two columns in the figure, corresponding to the "normal" application development process and quality goals, and the development of security requirements. It is assumed that no explicit activity is needed to elicit the organization's control principles, and these can therefore be fed

directly into the Identification of Security Goals activity.

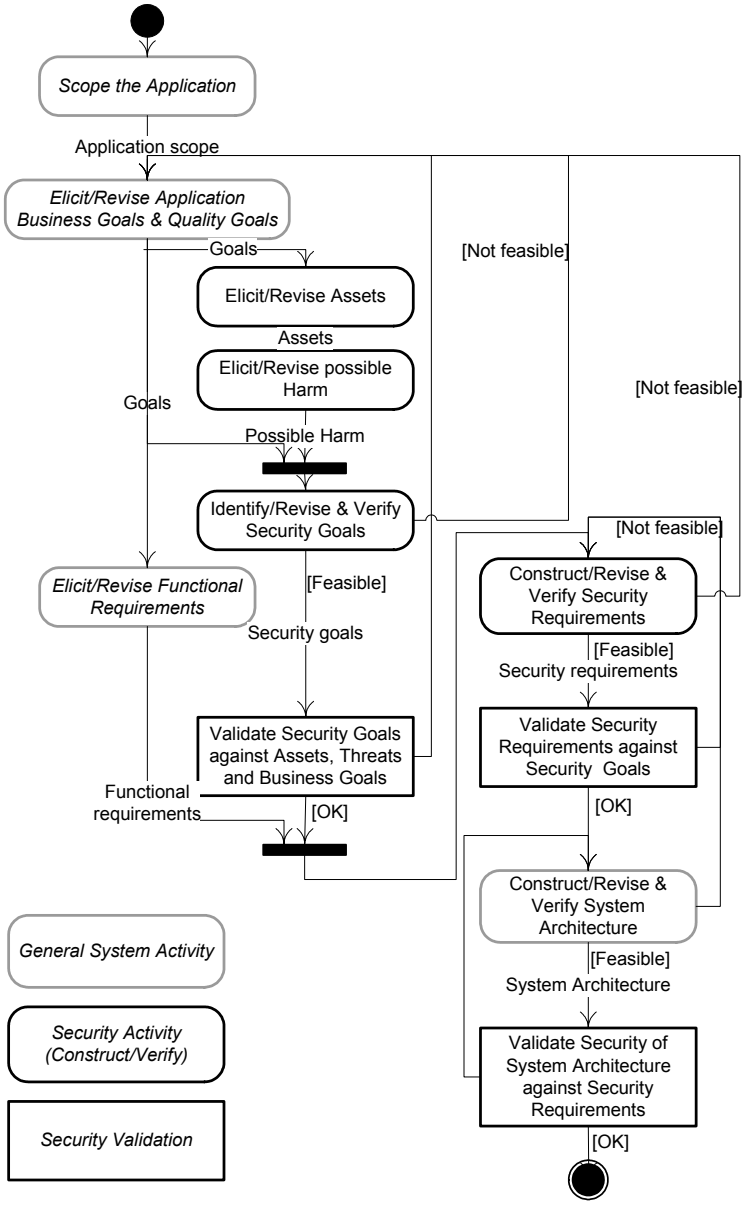

Figure 3 – Security Requirements Process Overview (Activity Diagram)

Lines coming out of the bottom of an activity box indicate the successful completion of an activity and carry with them a core artifact into the next activity. Lines coming out of the side of an activity box denote failure and imply the need to iterate back up the process in order to revise an earlier activity. Failure can be one of two kinds: it has been found to be infeasible to create a consistent set of the artifacts that are constructed by that activity; or validation of the artifacts

against a higher level – such as validation of security requirements against security goals – shows that they fail to meet their aims. This occurs if it has not been possible to construct a satisfactory satisfaction argument, or if a vulnerability has been found. The iteration may cascade upwards if the architecture is not feasible without a revision of the business or security goals.

Step: Identify Security Goals

The security goals of the system owner are derived from a combination of three different sources: the possible harm to assets; management control principles; and application business goals, which will determine the applicability of management control principles, for example by defining those privileges that are needed for the application prior to excluding those that are not.

Note that other legitimate stakeholders may have other security goals that conflict with these. The set of relevant security goals may be mutually inconsistent, and inconsistencies will need to be resolved during the goal analysis process, before a set of consistent requirements can be reached. On the other hand, the goals of attackers are not considered to be a part, even negated, of the security goals of the system. The goals of the system owner and other legitimate stakeholders are not directly related to the goals of attackers, because security is not a zero sum game like football. In football, the goals won by an attacker are exactly the goals lost by the defender. Security is different; there is no exact equivalence between the losses incurred by the asset owner and the gains of the attacker. To see this, look at two examples:

- Robert Morris unleashed the Internet Worm (Spafford, 1989), causing millions of dollars of damage, apparently as an experiment without serious malicious intent. The positive value to the attacker was much less than the loss incurred by the attacked sites.
- Many virus writers today are prepared to expend huge effort in writing a still more ingenious virus, which may cause little damage (screen message "You've got a Virus"). Generally, there

is no simple relationship between the gains of a virus writer and the losses incurred by those who are attacked.

The consequences of security not being a zero sum game are twofold: the evaluation of possible harm to an asset can generally be carried out without reference to particular attackers; and consideration of the goals of attackers cannot be used simply to arrive at the goals of a defender to prevent harm, i.e. their security goals.

Step: Identify Security Requirements

Recall that we define security requirements to be the constraints on functional requirements that are needed to achieve security goals. In the process, one determines which assets will be implicated in satisfying a particular functional requirement by drawing the context for that functional requirement as a problem diagram. The list of assets and the type of function will produce a list of threats that must be mitigated. The security requirements are these mitigations, constraining the function in ways that will achieve the security goals.

A simple example of such a constraint is:

```
The system shall provide Personnel Information only to members of Human Resources Dept.
```

The constraint ("`only to …`") is secondary to the function ("`provide Personnel Information`"); it only makes sense in the context of the function. One might also impose temporal constraints:

```
The system shall provide Personnel Information only during normal office hours;
```

and complex constraints on traces:

```
The system shall provide information about an organization only to any person who has not previously accessed information about a competitor organization (the Chinese Wall Security Policy, (Brewer & Nash, 1989)).
```

Availability requirements will need to express constraints on response time:

```
The system shall provide Personnel Information within 1 hour for 99% of requests.
```

We note that this differs only in magnitude from a Response Time quality goal, which might use the same format to require a sub-second response time.

Step: Construction of Satisfaction Arguments

The next step is to validate the security requirements against the system architecture by constructing the two-part security satisfaction arguments from a problem diagram and a set of security requirements. This step is described in the case study, below.

CASE STUDY

This case study, of a Personnel Information display system, is used to validate the framework that we have set out above and to bring out further issues for discussion. We first set out a simple system, with business goals, from which functional requirements are derived, and then show how the system security requirements are derived from the application of the organization's security goals to the functional requirements. Next, the satisfaction arguments are constructed. Given the system security requirements, there are design decisions to be made about where to locate the security functionality and the approach to be used, and one example of this is provided.

From System Business Goals to Security Requirements

A simple human resources application will be used in this section to illustrate the use of our process. We assume that the business goals have been elicited and that there is only one goal:

```
FG1: Provision of people's personnel information to them.
```

We further assume that initial requirements have been elicited and that there is only one functional requirement:

```
FREQ1: On request from a Person (member of People), the system shall display
personnel information (PersonInf) for a specified payroll number (Payroll#) to
that Person.
```

Further analysis shows that, ignoring physical assets such as the computers and the buildings, there is only one asset implicated in the system: PersonInf, an information asset.

The harms that involve PersonInf are exposure (loss of confidentiality), alteration (loss of

integrity), and denial of service (loss of availability). The first harm gives rise to the security goal SG1: prevent unauthorized exposure of PersonInf. Likewise, the second harm gives rise to the goal SG2: prevent unauthorized alteration of PersonInf, and the third harm gives rise to SG3: prevent denial of access to PersonInf by authorized persons.

The next step is to derive security requirements from the combination of functional goals and security goals. Remember that security requirements constrain the function called for by a functional goal. Applying SG1 to FREQ1, we derive the security requirement (constraint) SR1: Personnel information must be provided only to HR staff. We cannot apply SG2 to any functional requirement, because no requirement permits modification of PersonInf. Applying SG3 to FREQ1, we derive the (somewhat arbitrary) security requirement SR2: Personnel information must be provided to HR staff within 60 minutes of its request.

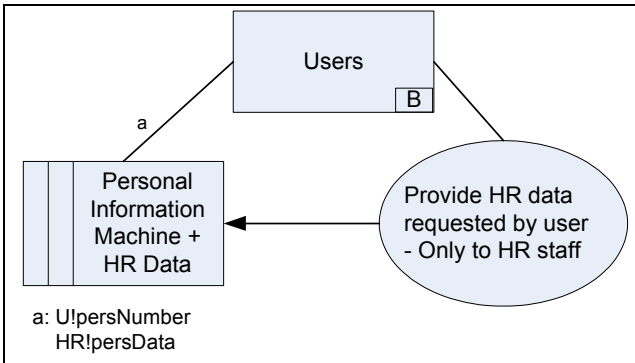

Figure 4 – Problem diagram for the HR data retrieval application

Although we have derived two security requirements, for reasons of space we will only look at one of them: SR1. Figure 4 shows the initial problem diagram for this application. There are two phenomena of interest. The first, U!persNumber, is the user's request for personnel information. The second, HR!persData, is the information returned by the request.

We begin by constructing an outer argument that proves the claim: HR data is provided only to HR staff.

Constructing Satisfaction Arguments

We wish to construct a convincing satisfaction argument that a system can satisfy its security requirements. The reader may note the use of the word "can", instead of the word "will". We use the phrase "can satisfy" because we do not know if the eventual implementation will respect the specifications. Nor do we know if the system will introduce unintended vulnerabilities, which manifest themselves as phenomena not described in the behavioral specification.

The Outer Argument

We first attempt to construct the outer argument for the HR problem shown in Figure 4. Recall that this argument will take the form

```
(domain property premises) ⊢ security requirement
```

There are two domains in the problem: the biddable domain 'users' and the machine (which contains the data). To construct the argument, we must first express the behavior of the system more formally, which we do using a notation based on the causal logic described in (Moffett, Hall, Coombes, & McDermid, 1996). In this logic, the behavior of the domains in Figure 4, expressed in terms of the phenomena, is:

```
1. U!persNum shall cause M!persData
```

A major problem is immediately exposed. Given what we see in the behavior description, there is no way to connect the system's behavior to the security requirement, as membership of the Users domain is not made apparent. No formal argument can be constructed. We must ask the system designers for help. There are (at least) three design choices:

1. Introduce some physical restriction, such as a guard, to ensure that the membership of the domain 'users' is restricted to HR staff. Doing so would permit construction of the following inner argument (proof):

```
M is defined as (User ∈ HR)
D is defined as (phenomenon HR!persData)
D → M  (if info is displayed, then user ∈ HR)
D      (info is displayed)
M      (therefore user ∈ HR)
```

2. Introduce phenomena into the system permitting authentication and authorization of a ‘user’.
3. Introduce a trust assumption (TA) asserting that the membership of ‘users’ is limited to HR staff, even though no information is available to support the assertion.

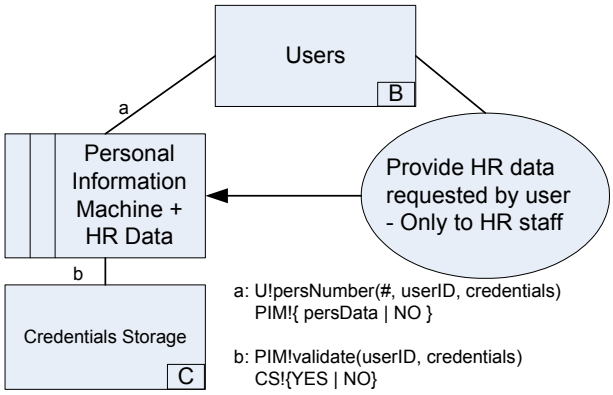

Figure 5 – New HR staff problem diagram

To make the example more interesting, we choose option 2. The resulting problem diagram is shown in Figure 5. The diagram shows that the user is to supply some sort of credentials along with the request for information. These credentials are passed to an external authentication and authorization engine, which answers yes or no. If the answer is yes, then the machine provides the data; otherwise, the data is refused. The corresponding behavior specification is:

```
1. U!(UserId, credentials, Payroll#) shall cause PIM!Validate(UserId, HR,
   credentials)
2. if isValid(UserId, credentials)
     PIM!Validate(HR, UserId, credentials) shall cause CS!YES
   else
     PIM!Validate(HR, UserId, credentials) shall cause CS!NO
3. CS!YES shall cause PIM!PersonInf(Payroll#)
4. CS!NO shall cause PIM!NO
```

We must now construct the satisfaction argument for the new ‘Users’ domain. We begin with the outer argument, first defining the symbols to be used. These are shown in the following table.

Symbol	Derived from (see Figure 5)
I : Input request	U!(UserId, credentials, Payroll#)
V: Validate Creds	PIM!Validate(HR, UserId, credentials)
Y: ReplyYes	CS!YES
D: DisplayInfo	PIM!PersonInf(Payroll#)
C: CredsAreValid	isValid(UserId, credentials)
M: MemberOfHR	Conclusion: user is member of HR

The following predicate logic premises are derived from the behavioral specification. These premises are the grounds used in the formal argument and, if necessary, will be supported by informal arguments.

Name	Premise	Description
P1	I → V	Input of request shall cause validation
P2	C → M	If credentials are valid then user is a member of HR
P3	Y → V&C	A Yes happens only if credentials are valid and validated
P4	D → Y	Display happens only if the answer was Yes

As the requirement is that we display information only to a member of HR, we include D as a premise and M as the conclusion. Thus we want to show:

```
(P1, P2, P3, P4, D ⊢ M).
```

A proof is shown in Figure 6.

1	I → V	(Premise)
2	C → M	(Premise)
3	Y → V & C	(Premise)
4	D → Y	(Premise)
5	D	(Premise)
6	Y	(Detach (→ elimination), 4, 5)
7	V & C	(Detach, 3, 6)
8	V	(Split (& elimination), 7)
9	C	(Split, 7)
10	M	(Detach, 2, 9)
11	D → M	(Conclusion, 5)

Figure 6 – Proof: the security argument is satisfied

The Inner Arguments

Each of the rules used in the outer argument should be examined critically. We begin with the premises P1, P3, & P4. These are probably not controversial, because one can say that they are

part of the specification of the system to be implemented. The arguments thus consist of one trust assumption, as shown in the following utterance in our argument language:

```
let G1 = "system will be correctly implemented";
given grounds G1 thus claim P1.
given grounds G1 thus claim P3.
given grounds G1 thus claim P4.
```

Premise P2 is more complex. This premise is making a claim about the membership of the domain ‘Users’ by saying that if a user has valid credentials, then that user must be a member of HR. An argument for this claim is shown below. This argument incorporates 3 trust assumptions: G2, G3, and G4.

```
given grounds
  G2: "Valid credentials are given only to HR members"
warranted by
(
 given grounds
  G3: "Credentials are given in person"
 warranted by
  G4: "Credential administrators are honest & reliable"
 thus claim
  C1: "Credential administration is correct"
)
thus claim
 P2: "HR credentials provided --> HR member"
rebutted by
 R1: "HR member is dishonest",
 R2: "social engineering attack succeeds",
 R3: "person keeps credentials when changing depts" .
```

The three rebuttals in the argument require some treatment. Recall that rebuttals express conditions under which the argument does not hold. If the rebuttals remain in the argument, they create implicit trust assumptions saying that the conditions expressed in the rebuttals will not occur, which may be acceptable. Alternatively, one could construct an argument against a rebuttal. If we assume that the stakeholder is unwilling to accept R1, then the system must somehow be changed to mitigate the rebuttal.

Removing Rebuttals by Adding Function

At times the most straightforward way to remove a rebuttal might be to add functionality to a system. The additional functionality would permit adding new grounds or warrants to mitigate

the conditions that permit the rebuttal.

As an example, consider R1: a dishonest HR member sells credentials. One could mitigate this risk by increasing the probability that an unusual use of the employee's credentials would be detected, thus raising the probability that the misuse would be detected. Doing so is probably most easily accomplished by adding new functionality to the system. In our example, we add two functional requirements to the system:

- FREQ2: all uses of HR credentials are logged
- FREQ3: any use of HR credentials from a location outside the HR department is immediately signaled by email to the HR director.

These functional requirements would then be used as grounds in an argument against the rebuttal R1:

```
given grounds
 G5: "uses of HR creds are logged (FREQ2)"
  and
 G6: "uses of HR creds from outside are emailed (FREQ3)"
warranted by
 G7: "these actions increase the probability of detecting improper use of
creds"
  and
 G8: "the employee does not want to get caught"
thus claim
 C2: "HR members will not sell their credentials".
```

C2 is added as a mitigating proposition to the rebuttal in argument 1 `(R1: "HR member is dishonest" mitigated by C2)`.

DISCUSSION AND FUTURE TRENDS

Constructing Inner Arguments

One question that arises is "how does the analyst find rebuttals, grounds, and warrants?" Unfortunately, we cannot propose a recipe. We suggest a method inspired by the how/why questions used in goal-oriented requirements engineering methods (e.g. KAOS (van Lamsweerde, 2001)). Given a claim, the analyst asks 'why is this claim true?' and 'what happens

if it is not true?' The analyst first to choose which claim is being argued, and then use the 'why' question to gather the grounds that are pertinent to the claim along with the warrants that connect the grounds to the claim. The argument is then constructed.

The analyst next asks the question "what can prevent this claim from being true?" The answers are the initial rebuttals. Some of these rebuttals will be challenges of the grounds or warrants; these create the need for sub-arguments where the challenged item is a claim. In other cases, the rebuttal will not be addressed, thereby creating an implicit trust assumption stating that the event(s) described in the rebuttal are not to be considered. A third possibility is to add new grounds to the argument that remove the conditions assumed by the rebuttal.

Problem vs. Solution Space

A reasonable objection to argumentation as described in this chapter is that we are designing the system in order to determine its requirements. To some extent, this is true; the domains included in the system are being more finely described iteratively.

However, we argue that the part of the system being constructed is the *machine*, and we are not designing that. By applying an iterative process that interleaves requirements and design (Nuseibeh, 2001), we are specifying the environment (or context) that the machine lives within. These specifications include additional domains that need to exist (perhaps inside the machine), and additional phenomena required to make use of these domains.

Security Functional Requirements

Adding functionality to support security requirements creates a traceability problem. This chapter provided two situations where this sort of functionality was added: addition of credential verification to permit the outer argument to be constructed, and addition of monitoring and logging functionality to support removal of the dishonest employee rebuttal. Somehow these

functions must remain connected with the security requirement they support, because the need for these functions could change or disappear if the security requirement changes.

Integration with Other RE Frameworks

One area for future work is to adapt the process described in this chapter to other requirements engineering frameworks such as KAOS (van Lamsweerde, 2001), *i** (Chung, Nixon, Yu, & Mylopoulos, 2000; Yu, 1997), and SCR (Heitmeyer, Kirby, Lebaw, & Bharadwaj, 1998). All of these frameworks are amenable to using the steps of our process, especially asset and harm analysis, and construction of satisfaction arguments. However, as each of the above frameworks specifies behavior in a very different way, the construction of the outer argument will be different, and these differences must be investigated.

The Role of Automation

The structure of the inner argument language permits some forms of automated analysis. Some options under consideration include generating lists of claims supporting a particular premise and testing of the argument through negation of particular trust assumptions. We are also considering using tools like Araucaria (Reed, 2005) for diagramming the arguments, generating the appropriate input for the program. We have already started using Compendium (Compendium Institute, 2005) to capture problem frame diagrams. We have used BlobLogic (Howitt, 2005) and DC Proof (Christensen, 2005) to help construct and check outer arguments, and we are investigating other proof construction aids.

CONCLUSIONS

At the beginning of this chapter, we stated four motivating concerns, repeated here:

- The "what" of security requirements – its core artifacts – must be understood before the

"how" of their construction and analysis.

- Security cannot be considered as a feature of software alone; it is concerned with the prevention of harm in the real world. We must therefore consider the security requirements of real-world systems in addition to the software.
- Since security is largely concerned with prevention of misuse of system functions, security requirements can most usefully be defined by considering them as constraints upon functional requirements.
- Since security is by definition an 'open world' problem (the domain of analysis will always be too small), any argument that a system will satisfy its security requirements must take non-provable assertions about the real world into account.

In this chapter we have addressed these concerns by presenting a precise definition of security requirements, a framework for determining these requirements, and a structure for arguing that the requirements will be satisfied. Other advantages of our approach are:

- Security requirements are naturally integrated with the system's functional requirements and with constraints derived from other sources. An integrated development is possible.
- This has the consequence that interactions and trade-offs between security and other quality requirements can be analyzed. For example, interactions and trade-offs between them can be considered in terms of the different required constraints on the same functional requirements.
- The two-level satisfaction arguments facilitate showing that a system can meet its security requirements. The structure behind the arguments assists in finding system-level vulnerabilities. By first requiring the construction of the formal argument based on domain properties, one discovers which domain properties are critical for security. Constructing the

informal argument showing that these domain properties can be trusted helps point the analyst toward vulnerabilities; the rebuttal is an important part of this process. Vulnerabilities found in this way are removed either through modification of the problem, addition of security functional requirements, or through addition of trust assumptions that explain why the vulnerability can be discounted.

We claim that this framework will help requirements and security engineers to understand the place of the various synthetic and analytical activities that have previously been carried out in isolation. The framework has raised a number of open issues, mentioned in the discussion, but we believe that it provides a way forward to effective co-operation between the two disciplines of requirements and security.

Acknowledgements:

The authors wish to thank Michael Jackson for his continuous involvement and support. Thanks also to Simon Buckingham Shum for many helpful conversations about argumentation. The financial support of the Leverhulme Trust and the Royal Academy of Engineering is gratefully acknowledged, as is EU support of the E-LeGI project, number IST-002205.

References:

Alexander, I. (2003). Misuse Cases in Systems Engineering. *Computing and Control Engineering Journal, 14*(1), 40-45.

Anderson, R. (1996). A Security Policy Model for Clinical Information Systems. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy* (pp. 30-43). Oakland CA USA.

Anderson, R. (2001). *Security Engineering: A Guide to Building Dependable Distributed Systems*.

Antón, A. I., & Earp, J. B. (2001). Strategies for Developing Policies and Requirements for Secure E-Commerce Systems. In A. K. Ghosh (Ed.), *E-Commerce Security and Privacy* (Vol. 2, pp. 29-46): Kluwer Academic Publishers.

Attwood, K., Kelly, T., & McDermid, J. (2004). The Use of Satisfaction Arguments for Traceability in Requirements Reuse for System Families: Position Paper. In *Proceedings of the International Workshop on Requirements Reuse in System Family Engineering, Eighth International Conference on Software Reuse* (pp. 18-21). Carlos III University of Madrid, Madrid Spain.

Brewer, D. F. C., & Nash, M. J. (1989). The Chinese Wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy* (pp. 206 - 214). Oakland CA USA: IEEE Computer Society Press.

Buckingham Shum, S. J. (2003). The Roots of Computer Supported Argument Visualization. In P. A. Kirschner, S. J. Buckingham Shum & C. S. Carr (Eds.), *Visualizing Argumentation: Software Tools for Collaborative and Educational Sense-Making* (pp. 3-24). London UK: Springer-Verlag.

Burge, J. E., & Brown, D. C. (2004). An Integrated Approach for Software Design Checking Using Design Rationale. In J. S. Gero (Ed.), *Proceedings of the First International Conference on Design Computing and Cognition* (pp. 557-576). Cambridge MA USA: Kluwer Academic Press.

Christensen, D. (2005). *DC Proof*. Retrieved 9 Nov, 2005, from http://www.dcproof.com/

Chung, L., Nixon, B., Yu, E., & Mylopoulos, J. (2000). *Non-Functional Requirements in Software Engineering*: Kluwer Academic Publishers.

Compendium Institute. (2005). *Compendium*, 2005, from http://www.compendiuminstitute.org/

Devanbu, P., & Stubblebine, S. (2000). Software Engineering for Security: A Roadmap. In A. Finkelstein (Ed.), *The Future of Software Engineering*: ACM Press.

Finkelstein, A., & Fuks, H. (1989). Multiparty Specification. In *Proceedings of the 5th International Workshop on Software Specification and Design* (pp. 185-195). Pittsburgh PA USA.

Fischer, G., Lemke, A. C., McCall, R., & Morch, A. (1996). Making Argumentation Serve Design. In T. Moran & J. Carroll (Eds.), *Design Rationale Concepts, Techniques, and Use* (pp. 267-293): Lawrence Erlbaum and Associates.

Haley, C. B., Laney, R. C., Moffett, J. D., & Nuseibeh, B. (2004). The Effect of Trust Assumptions on the Elaboration of Security Requirements. In *Proceedings of the 12th International Requirements Engineering Conference (RE'04)* (pp. 102-111). Kyoto Japan: IEEE Computer Society Press.

Haley, C. B., Laney, R. C., & Nuseibeh, B. (2004). Deriving Security Requirements from Crosscutting Threat Descriptions. In *Proceedings of the Third International Conference on Aspect-Oriented Software Development (AOSD'04)* (pp. 112-121). Lancaster UK: ACM Press.

Haley, C. B., Moffett, J. D., Laney, R., & Nuseibeh, B. (2005). Arguing Security: Validating Security Requirements

Using Structured Argumentation. In *Proceedings of the Third Symposium on Requirements Engineering for Information Security (SREIS'05) held in conjunction with the 13th International Requirements Engineering Conference (RE'05)*. Paris, France.

Heitmeyer, C. L. (2001). Applying 'Practical' Formal Methods to the Specification and Analysis of Security Properties. In *Proceedings of the International Workshop on Information Assurance in Computer Networks: Methods, Models, and Architectures for Network Computer Security (MMM ACNS 2001)* (Vol. 2052, pp. 84-89). St. Petersburg, Russia: Springer-Verlag Heidelberg.

Heitmeyer, C. L., Kirby, J., Lebaw, B. G., & Bharadwaj, R. (1998). SCR*: A Toolset for Specifying and Analyzing Software Requirements. In A. J. Hu & M. Y. Vardi (Eds.), *Proceedings of the 10th International Conference on Computer Aided Verification* (Vol. 1427, pp. 526-531). Vancouver BC Canada: Springer.

Howard, M., & LeBlanc, D. (2001). *Writing Secure Code*: Microsoft Press.

Howitt, C. (2005). *BlobLogic*. Retrieved 9 Nov, 2005, from http://users.ox.ac.uk/~univ0675/blob/blobSplash.html

ISO/IEC. (1999). *Information Technology - Security Techniques - Evaluation Criteria for IT Security - Part 1: Introduction and General Model* (International Standard No. 15408-1). Geneva Switzerland: ISO/IEC.

Jackson, M. (2001). *Problem Frames*: Addison Wesley.

Kelly, T. P. (1999). *Arguing Safety - A Systematic Approach to Safety Case Management.* Unpublished D.Phil Dissertation, University of York, York.

Kotonya, G., & Sommerville, I. (1998). *Requirements Engineering: Processes and Techniques*. United Kingdom: John Wiley and Sons.

van Lamsweerde, A. (2001). Goal-oriented Requirements Engineering: A Guided Tour. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering (RE'01)* (pp. 249-263). Toronto, Canada: IEEE Computer Society Press.

van Lamsweerde, A. (2004). Elaborating Security Requirements by Construction of Intentional Anti-Models. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)* (pp. 148-157). Edinburgh Scotland.

Lee, J., & Lai, K. Y. (1991). What's in Design Rationale? *Human-Computer Interaction - Special Issue on Design Rationale, 6*(3-4), 251-280.

Lee, Y., Lee, J., & Lee, Z. (2002). Integrating Software Lifecycle Process Standards with Security Engineering. *Computers and Security, 21*(4), 345-355.

Liu, L., Yu, E., & Mylopoulos, J. (2003). Security and Privacy Requirements Analysis Within a Social Setting. In *Proceedings of the 11th IEEE International Requirements Engineering Conference (RE'03)*. Monteray Bay, CA USA.

McDermott, J., & Fox, C. (1999). Using Abuse Case Models for Security Requirements Analysis. In *Proceedings of the 15th Computer Security Applications Conference (ACSAC'99)* (pp. 55-64). Phoenix AZ USA: IEEE Computer Society Press.

Moffett, J. D., Haley, C. B., & Nuseibeh, B. (2004). *Core Security Requirements Artefacts* (Technical Report No. 2004/23). Milton Keynes UK: Department of Computing, The Open University.

Moffett, J. D., Hall, J. G., Coombes, A., & McDermid, J. A. (1996). A Model for a Causal Logic for Requirements Engineering. *Requirements Engineering, 1*(1), 27-46.

Mouratidis, H., Giorgini, P., & Manson, G. (2003). Integrating Security and Systems Engineering: Toward the Modelling of Secure Information Systems. In *Proceedings of the The 15th Conference on Advanced Information Systems Engineering (CAiSE'03)*. Klagenfurt/Velden, Austria: Springer-Verlag.

Mylopoulos, J., Borgida, A., Jarke, M., & Koubarakis, M. (1990). Telos: Representing Knowledge About Information Systems. *ACM Transactions on Information Systems (TOIS), 8*(4), 325 - 362.

Newman, S. E., & Marshall, C. C. (1991). *Pushing Toulmin Too Far: Learning From an Argument Representation Scheme* (Technical Report No. SSL-92-45). Palo Alto CA USA: Xerox PARC.

Nuseibeh, B. (2001). Weaving Together Requirements and Architectures. *Computer (IEEE), 34*(3), 115-117.

Potts, C., & Bruns, G. (1988). Recording the Reasons for Design Decisions. In *Proceedings of the 10th International Conference on Software Engineering (ICSE'88)* (pp. 418-427). Singapore: IEEE Computer Society.

Ramesh, B., & Dhar, V. (1992). Supporting Systems Development by Capturing Deliberations During Requirements Engineering. *IEEE Transactions on Software Engineering, 18*(6), 498-510.

Reed, C. (2005). *Araucaria*, from http://araucaria.computing.dundee.ac.uk/

Rushby, J. (2001). Security Requirements Specifications: How and What? In *Proceedings of the Symposium on Requirements Engineering for Information Security (SREIS)*. Indianapolis, IN USA.

Sindre, G., & Opdahl, A. L. (2000). Eliciting Security Requirements by Misuse Cases. In *Proceedings of the 37th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS-Pacific'00)* (pp. 120-131). Sydney Australia.

Spafford, E. H. (1989). The internet worm program: an analysis. *ACM SIGCOMM Computer Communication Review, 19*(1), 17-57.

Tettero, O., Out, D. J., Franken, H. M., & Schot, J. (1997). Information security embedded in the design of telematics systems. *Computers and Security, 16*(2), 145-164.

Toulmin, S. E. (1958). *The Uses of Argument*. Cambridge: Cambridge University Press.

Toulmin, S. E., Rieke, R. D., & Janik, A. (1979). *An Introduction to Reasoning*. New York: Macmillan.

Yu, E. (1997). Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering. In *Proceedings of the Third IEEE International Symposium on Requirements Engineering (RE'97)* (pp. 226-235). Annapolis MD USA.

Zave, P., & Jackson, M. (1997). Four Dark Corners of Requirements Engineering. *Transactions on Software Engineering and Methodology (ACM), 6*(1), 1-30.